Arrive Technologies Inc.

Guidelines

ATVN

FPGA/ASIC Design Guideline

Abstract: This document provides basic FPGA/ASIC design guidelines for AISC engineers working at Arrive Technologies VietNam

Revision:	***1.1***
Last Updated:	***03-Aug-06***

Author:

– AT ASIC Design Group

Document History

Revision	Date	Description
1.0	Aug 2003	Document created
1.1	Aug 2006	New Template updated

Contents

Figures

1. Introduction

This document outlines the guidelines and conventions used by Arrive Technologies. They are aimed at the design of ASIC/FPGA by Verilog HDL.

2. Organization of Design Data

Most of the design data is stored in a main directory at the ATVN data server. This includes the Verilog source code of the design, design documents, references, scripts used for simulation and synthesis and configuration files. Some common script files may be stored in a global directory.

3. Coding Conventions

3.1. Coding Organization

The Verilog code is stored in an own directory; synthesis and layout data is stored in others. The following guidelines should be followed:

- Each modules gets its own file which should have the same name as the module plus the suffix ".v"
- Numerical constants should be defined symbolically by a **`define** statement. This help to reduce redundancy. If you change one parameter of your design you shouldn't have to change your code at more than one location
- Use **parameter** statements for modules where you need the same module with the same logic but only different parameters like bus widths etc. This also helps to minimize redundancy

3.2. Coding Formatting

See Verilog Coding Standard Document.

3.3. Design Partitioning

Partition the design into small functional blocks, and use a behavioral style for each block. Avoid gate level descriptions except for critical parts of the design.

The top and core level of the RTL should not have any logic in it.

It is recommended that the I/O buffers be kept together, separate from any design logic. If possible, the I/O buffers should be placed at the top level of the design.

Keeping the I/O separate will assist in extracting just the core logic as FPGA I/O buffers must be replaced by their ASIC counterparts. I/O buffers by their nature cannot be synthesized and hence need special treatment. It is difficult and error prone to chase I/O buffers throughout a design hierarchy.

A common and acceptable practice is to place all the logic descriptions under a "core" module and instantiate all the I/O buffers along with the single core instance inside the top-level module. In this manner, when migrating to an ASIC, a new top-level module can be built instantiating ASIC I/O buffers.

3.4. Circuit Concepts

Despite the formal conventions making it easier to read modules and to interface to other modules there are conventions referring to the type of circuit that is described by the Verilog code. These try to avoid some pitfalls, make it easier to test the design in a stringent way and try to ensure the simulations correspond to the behavior of the real chip as far as possible.

- Make a synchronous design as far as possible. Many problems can be avoided if you ensure that you have a common clock with the same phase for all sequential elements. Use positive edge of this clock to operate all flip-flops in your entire design

- If you have to use an asynchronous element in your design, document the reason for using it and document the constraints it poses on the surrounding circuit. Encapsulate it in a synchronous module if possible
- Ensure that after power-up or reset the system is in a stable state. No activity should occur without changing the input signals

4. Synchronous Design Techniques

Some synchronous design suggestions:

- Use a single master clock
- Use a single master set or reset. Preferably, use asynchronous resets because they work independently from the clock. When an asynchronous reset establishes the initial state, it puts the entire circuit into a known state and helps make logic simulation and manufacturing test easier
- Avoid race conditions on de-asserting concurrent set and preset signals. You cannot predict in simulation how the flop will behave when both set and reset are de-asserted close in time
- Use clock-enabled flip-flops. Do not use gated clocks (gated clocks require special attention. Be sure to design adequate margins and include comments). Synthesizing the gated clocks is dangerous. This configuration is very sensitive to glitches and simultaneous switching inputs, and may even delay the clock. Further in the design process, problems with testability may also occur
- Use clock-enabled flip-flops for clock division. In many FPGA implementations, ripple clock dividers are popular. Not only can ripple clock dividers cause problems with EDA (Electronic Design Automation) tools, the generated clock will experience a phase delay
- Use clock-enabled flip-flops to avoid glitching state decoders. FPGAs are sometimes tolerant when a state decoder goes through “11” while changing from “01” to “10.” To ASICs, this causes implementation-dependent glitches. Using clock-enabled flip-flops not only avoids glitches but also adds no additional clock delays
- Have resets and transition states for Finite State Machines. Although FSMs are usually synchronous, they still can have issues during the conversion process. Make sure there are no dead states because during power-up the FSM can enter an unused state. Make sure reset is also available on your FSM to make life easier during simulation and test vector generation
- Synchronize all asynchronous inputs in your FSM. When input data and clock change at the same time, setup and hold time requirements may cause metastability problems
- Avoid latches; use flip-flops instead. Latches cause complications with static timing and timing-driven layout tools. Latches are difficult to analyze, and the gate savings between a latch and a flop are less important with submicron technology
- Watch out for incompletely specified logic combinations in “case” and “if-then-else” situations. These will infer a latch in an ASIC to exactly meet the HDL behavior
- Do not use combinational feedback loops. Setup and hold times cannot be determined by logic timing analysis or simulation
- Avoid the use of three-state logic in the core of your design. Use muxes instead
- Do not use clocks generated from combinational feedback circuits or logic. When clocks are generated by feeding clocks through some gates in the FPGA, the frequency will change during the ASIC conversion. The timing changes because the technology inherent in the ASIC is different from that in the FPGA

5. Synchronization Concepts

5.1. Single Signal Synchronization

5.1.1. Level Detection of Asynchronous Single Signal

We must use at least 2 flip-flops to prevent **metastability**.

Example: data_i is asynchronous single signal to clk

```
reg     meta;
reg     data_o;
always @(posedge clk or negedge rst_)
        if (!rst_)
                begin
                meta    <= 1’b0;
                data_o  <= 1’b0;
                end
        else
                begin
                meta    <= data_i;
                data_o  <= meta;
                end
```

5.1.2. Edge Detection of Asynchronous Single Signal

We must use at least 3 flip-flops to prevent metastability. Why 3 flip-flops? Because we use 2 flip-flops to prevent metastability and after that this corrected signal must be delayed one clock to detect edge (low to high or high to low). Here is the sample:

```
reg     meta;
reg     data_o;
reg     data_o1;        // delayed signal of data_o
always @(posedge clk or negedge rst_)
        if (!rstn)
                begin
                meta    <= 1’b0;
                data_o  <= 1’b0;
                data_o1 <= 1’b0;
                end
        else
                begin
                meta    <= data_i;
                data_o  <= meta;
                data_o1 <= data_o;
                end
wire    data_posedge;
wire    data_negedge;
assign data_posedge = ~data_o1 & data_o;     // detect posedge of data_i
assign data_negedge = data_o1 & ~data_o;     // detect negedge of data_i
```

5.1.3. Synchronization Pitfall

Never synchronize the same signal in multiple places! **Inconsistency will result**! (Due to placement)

5.2. Bus Synchronization

Note: never use single signal synchronizers on each bit.

We need a single point of synchronization for the entire bus. Bus synchronization can be done in two ways: handshaking and FIFO. Handshaking is used when the received bus signal is at low speed (for example CPU bus interface). When the received bus signal is at high speed, we must use FIFO (see FIFO design guideline for more information).

5.2.1. 4-phase Handshaking

This method is used in asynchronous system. It detects level of control signal.

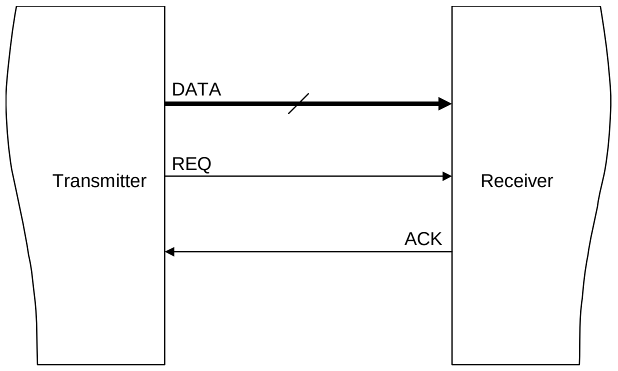

Figure 1: Four-phase Handshaking

Here is the procedure of 4-phase handshaking:

- Transmitter outputs DATA and then asserts REQ
- Receiver detects level of asynchronous single signal REQ. When it detects REQ = 1, it latches DATA and then asserts ACK
- Transmitter detects level of asynchronous single signal ACK. When it detects ACK = 1, it deasserts REQ, will not reassert it until ACK deasserts
- Receiver sees REQ deasserted, deasserts ACK when ready to continue

5.2.2. 2-phase Handshaking (alternate handshaking scheme)

This scheme is the same with 4-phase handshaking scheme except that edge of control signal detected.

- Transmitter outputs DATA and then changes state of REQ, will not change state of REQ again until after ACK changes state
- Receiver latches DATA. Once receiver is ready for more it changes state of ACK

5.2.3. Handshaking without ACK (alternate handshaking scheme)

This scheme is the same with 2-phase handshaking scheme except that there is no need of ACK signal.

Here is the procedure of this handshaking method:

- Transmitter outputs DATA and then asserts DATA_VALID.
- Receiver detects level of asynchronous single signal DATA_VALID. When it detects DATA_VALID = 1, it latches DATA.

Note: For back to back application, DATA_VALID must be a Manchester signal.

6. Other Design Concepts

6.1. Creating Safe State Machine

Safe State Machines

If the number of states (N) is a power of 2 and you use a binary or gray-code encoding algorithm, then the state machine is "safe." This ensures that you have M number of registers where N = 2M. Because all of the possible state values (or register statuses) are reachable, the design is "safe."

Unsafe State Machines

If the number of states is not a power of 2, or if you do not use binary or gray-code encoding algorithm (e.g., one-hot), then the state machine is "unsafe."

Generally, states that are not defined are covered by the "default" (for Verilog) branch of the case statement. The default operation of the synthesis tool will optimize away unreachable states in order to get a high-performance circuit. However, the optimization will create an "unsafe" circuit.

6.2. Pipeline

Refer to fflopx.v and fflopxe.v in macro directory.

6.3. FIFO Design Guideline

- For array memory FIFO, refer to fifox.v in macro directory
- For FIFO control only (flexible memory), refer to fifoctrlx.v in macro directory
- For FIFO clock converter (flexible memory), refer to ffclkconvx.v in macro directory.

6.4. RAM Interface

Refer to memspx.v or memrwx.v in macro directory.

6.5. CPU Interface

Global Processor Interface <--------> Local Processor Interface

```
upen    ---> // processor enable, active high. May be Pen1, Pen2, Pen3 if necessary
upa     ---> // address bus, the wide depending on number of local locations
updi    ---> // data in bus
updo    <--- // data out bus
```

if using cpu clock (frequency of local clocks is less than cpu clock)

```
upclk   ---> // cpu clk
upws    ---> // write strobe synchronized with pclk
uprs    <--- // read strobe synchronized with pclk
upack   <--- // acknowledge synchronized with pclk
```

if using local clock (frequency of local clocks is greater than cpu clock)

```
lclk    ---> // local clk
lws     ---> // write strobe synchronized with local clk
lrs     ---> // read strobe synchronized with local clk
lack    <--- // acknowledge synchronized with local clk
```

Make cpu write cycles and cpu read cycles as fast as possible. Should be less than about 10 cycles at cpu clock.

Another consideration, try to implement local processor interfaces and engines separately if possible

```
<Global PI> <-------------> <Local PI> <--------------> <Engine>
(cpu clk)           (local clk or cpu clk)            (local clk)
```

6.6. CPU Register

For configuration register, refer to pconfigx.v in macro directory

For sticky register, refer to stickyx.v in macro directory.

7. Simulation

Simulations take place at various stages of the design flow. They all use common test benches and stimulus vectors (test cases). The results are compared to make sure that the simulations consistent for all stages of the design.

The first stage is functional simulation of the Verilog code without timing information. This is used to achive logical correct behavior of the design. At later stage, the simulation is repeated with additional timing information of schematic and layout. The final simulation includes all delays of standard cells and nets with final placement and routing after layout.

7.1. Test Bench

Simulation of Verilog modules requires a test bench, which provides the environment of the module, like input signals or models of surrounding circuits. It can also contain modules to analyze and check the output of the module under test.

7.2. Test Case

Each user can define his own stimulus vector (test case) to test a function of the whole design.

7.3. Functional Simulation

The design entry is done with Verilog. The NC-Verilog simulator from Cadence is used to simulate the Verilog code. Simulation results can be displayed under the use of waveform display or test display. To avoid a huge dump file, it is recommended not to dump all signals of the design.

8. Test strategies

It is important to simulate all components and the whole design carefully and thoroughly in order to ensure that the chip will meet all requirements. This includes functional simulation, simulation with gate and path delays and analysis of all aspects not covered by simulation. Another important topic is the test of the chip after production.

8.1. Simulation

Simulation is done in three stages. First stage is functional simulation, which is used to debug the Verilog code and to ensure that the logical function of the description is correct and meets the requirements. The second stage is simulation after synthesis, which verifies that the synthesized circuit is equivalent to the functional description. The third stage is simulation with full delays, which is used to verify that no timing problems arise and the circuit meets the timing requirements.

The following aspects have to be considered for simulation:

- The test vectors should cover as many states of the circuit as possible. Special case should be taken for simulation of typical conditions under which the chip will be operated, boundary conditions like minimum and maximum values of counters, adder and multipliers, power-up… and some sort of arbitrary or random input to find unexpected errors
- Compare the results of the simulation to the results obtained independently. Avoid making the same error twice, one in the code you simulate and another time in the interpretation of your result. To prevent this, it is useful if the simulation is repeated by another person than the designer of the block under simulation
- All three stages of simulation should be done with the same test benches and vectors. The results must match
- Simulation results should be documented
- Simulation all parts of the complete design even trivial blocks

8.2. Scan Test

- Scan flip-flops in your design (they must be either all single-clock types or all dual-clock types, depending upon the chosen scan test methodology)
- Allocate one or several I/O pins for test purposes
- Avoid combinational feedback loops
- Avoid latches
- Avoid internal tristate buses
- Be sure that all clock signals for flip-flops are controllable directly from external clock input pins during test mode

- Be sure that all set and reset signals for flip-flops are fully controllable from external I/O pins during test mode